VIA PadLock Advanced Cryptography Engine

Programming Guide

Version 1.32



This is **Version 1.32** of the VIA PadLock Advanced Cryptography Engine Programming Guide.

© 2003 - 2005 VIA Technologies, Inc. All Rights Reserved.
© 2003 - 2005 Centaur Technology, Inc. All Rights Reserved.

VIA Technologies and Centaur Technology reserve the right to make changes in its products without notice in order to improve design or performance characteristics.

This publication neither states nor implies any representations or warranties of any kind, including but not limited to any implied warranty of merchantability or fitness for a particular purpose. No license, express or implied, to any intellectual property rights is granted by this document.

VIA Technologies and Centaur Technology make no representations or warranties with respect to the accuracy or completeness of the contents of this publication or the information contained herein, and reserves the right to make changes at any time, without notice. VIA Technologies and Centaur Technology disclaim responsibility for any consequences resulting from the use of the information included herein.

VIA and VIA C3 are trademarks of VIA Technologies, Inc.

CentaurHauls is a trademark of Centaur Technology, Inc..

Intel is a registered trademark of Intel Corporation.

Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

LIFE SUPPORT POLICY

VIA processor products are not authorized for use as components in life support or other medical devices or systems (hereinafter called life support devices) unless a specific written agreement pertaining to such intended use is executed between the manufacturer and an officer of VIA.

1. Life support devices are devices which (a) are intended for surgical implant into the body or (b) support or sustain life and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. This policy covers any component of a life support device or system whose failure to perform can cause the failure of the life support device or system, or to affect its safety or effectiveness.

Table of Contents

1 INTRODUCTION....2

2 ARCHITECTURE....3

2.1HARDWARE....3
2.2SOFTWARE....7
2.3MULTI-TASKING....9
2.4EXTENDED KEYS....11
2.5XCRYPTECB: ELECTRONIC CODE BOOK (ECB) MODE....12
2.6XCRYPTCBC: CIPHER BLOCK CHAINING (CBC) MODE....14
2.7XCRYPTCFB: CIPHER FEEDBACK (CFB) MODE....16
2.8XCRYPTOFB: OUTPUT FEEDBACK (OFB) MODE....19
2.9XCRYPTCTR: COUNTER (CTR) MODE....21

3 PROGRAMMING DETAILS....24

3.1MEMORY FORMATS....24
3.2CONTROL WORD POINTER....24
3.3DATA REGISTERS....26
3.4REP PREFIX....27
3.5EXCEPTION HANDLING....28

4 USING PADLOCK ACE....29

4.1ENABLING....29
4.2STARTUP....30
4.3MANUFACTURED STATE....31
4.4VERIFICATION....31

5 INSTRUCTION TIMING....32

5.1TIMING BASICS....32
5.2MEASURED PERFORMANCE....34

6 SAMPLE CODE....37

CHAPTER

1

INTRODUCTION

VIA provides a suite of security technologies called PadLock in all new processors, beginning with PadLock Random Number Generator in the VIA Nehemiah processor.

With stepping 8 of the Nehemiah processor, VIA added PadLock Advanced Cryptography Engine (PadLock ACE), a hardware implementation of the cryptographic algorithms of the Advanced Encryption Standard (AES). This programming guide documents PadLock ACE as implemented in both the VIA Nehemiah processor and the VIA Esther processor.

Separate Programming Guides are available for PadLock Random Number Generator (RNG), PadLock Montgomery Multiplier (PMM), and PadLock Hash Engine (PHE).

The CPUID identification of the VIA Esther processor is:

Vendor ID: CentaurHauls
Type: 0
Family: 6
Model: 10

The CPUID identification of VIA Nehemiah processors that support ACE is:

Vendor ID: CentaurHauls
Type: 0
Family: 6
Model: 9
Stepping: 8

For support, developers should contact ace_support@centtech.com.

CHAPTER

2

ARCHITECTURE

2.1 HARDWARE

PadLock Advanced Cryptography Engine (ACE) is implemented in hardware with associated microcode. The hardware performs the encryption functions, and the microcode provides a well-designed x86 instruction interface providing transparent multitasking. This section describes the PadLock ACE hardware structure. This information is hidden from the x86 instruction interface and is not necessary for proper use of PadLock ACE.

Figure 1 illustrates the conceptual architecture of PadLock ACE, a 128-bit wide unit added in parallel with the SSE unit sharing instruction, load, and store buses. PadLock ACE has a discrete set of internal registers that are not visible to x86 instructions. PadLock ACE instructions are issued, in order, to the SSE unit along with SSE instructions.

Once a PadLock ACE instruction is ready for execution, it is dispatched to the ACE hardware by the SSE unit. Instruction execution in other units, such as integer, MMX, and floating point, can proceed in parallel with ACE operations. SSE instructions, however, are blocked from execution until the ACE instruction completes, because all ACE operations end with storing data to memory. This store must be completed before subsequent SSE instructions can execute.

Figure 2 provides a further conceptual breakout of PadLock ACE components. There are four input registers and two output registers. In addition, there is a 128-bit wide, 16-entry RAM that holds the extended keys for encryption. This RAM can be loaded from memory, or by an extended key generation unit that calculates the extended key sequence based on the primary key. The AES encryption engine is pipelined and can operate on two 128-bit data blocks concurrently.

Figure 1. Adding Security Features to x86 Architecture

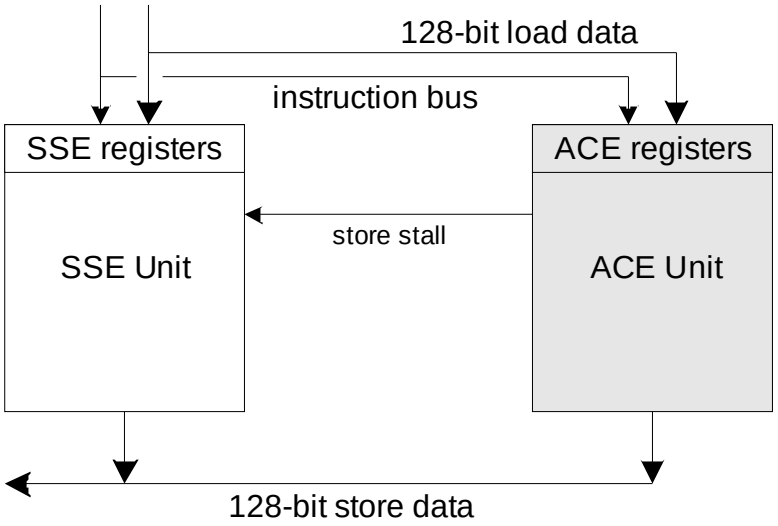
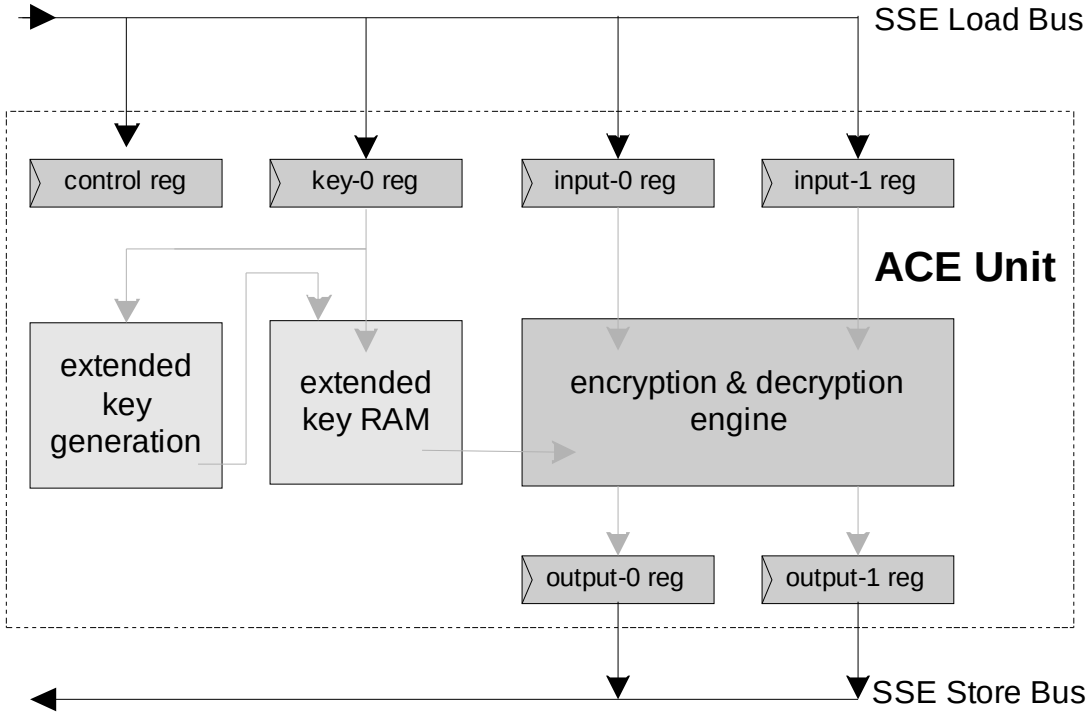
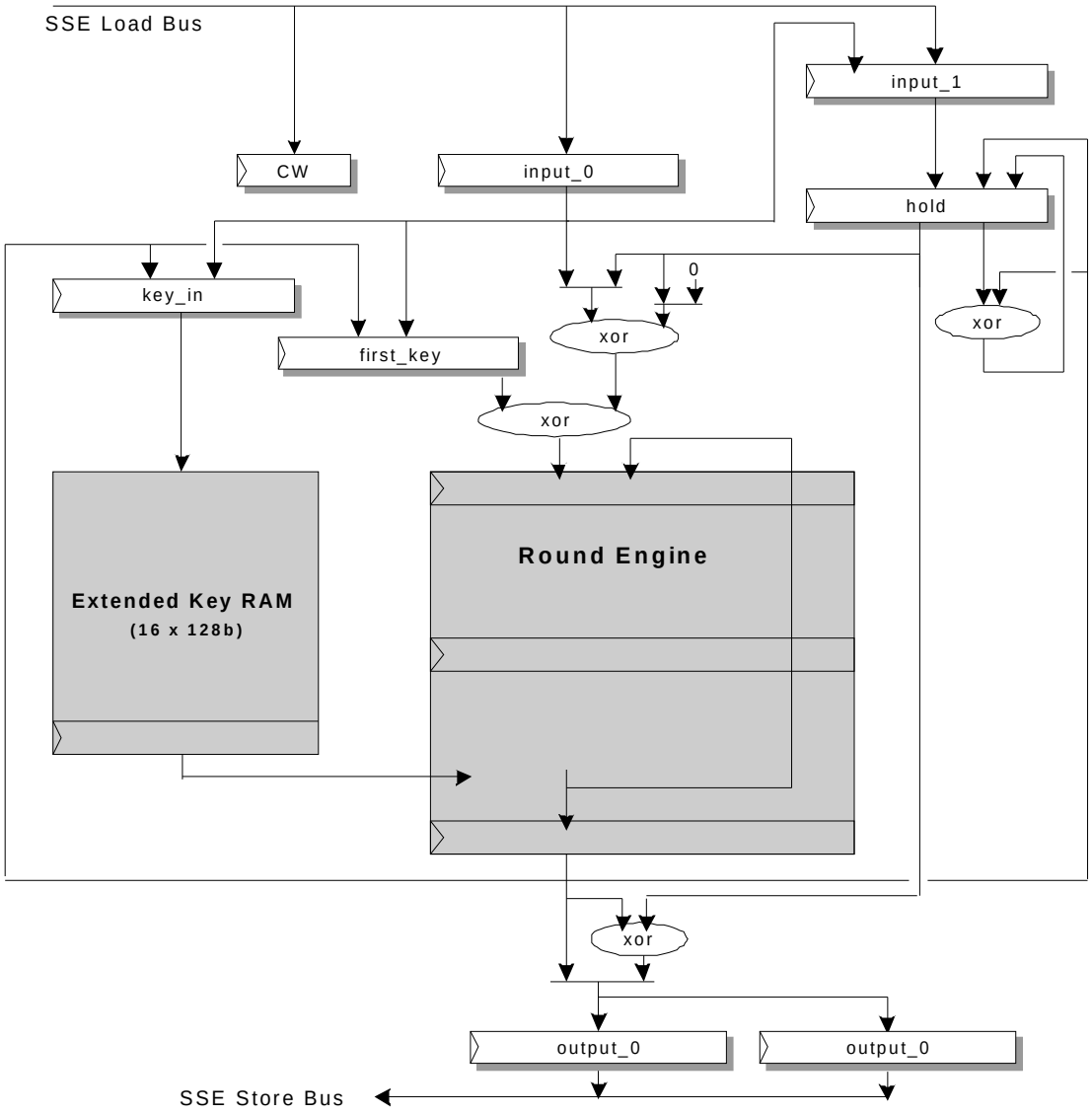


Figure 2. Conceptual ACE Hardware Overview



PadLock ACE is more complicated than shown in Figure 2 because ACE hardware directly supports several different operating modes. In addition, the extended-key generation hardware is not a distinct component, but rather is built into the encryption logic to minimize distinct circuitry. Figure 3 summarizes the hardware structure surrounding the round engine, which implements the AES encryption function.

Figure 3. ACE Hardware



Consider the data path for a typical instruction (XCRYPTECB). If key loading is necessary, microcode directs the loading of the key and extended keys are generated into or loaded into the extended key RAM. The RAM is large enough to hold an extended key for 15 rounds plus the initial key. The hardware automatically accesses the appropriate 128-bit extended key sequence needed for each round.

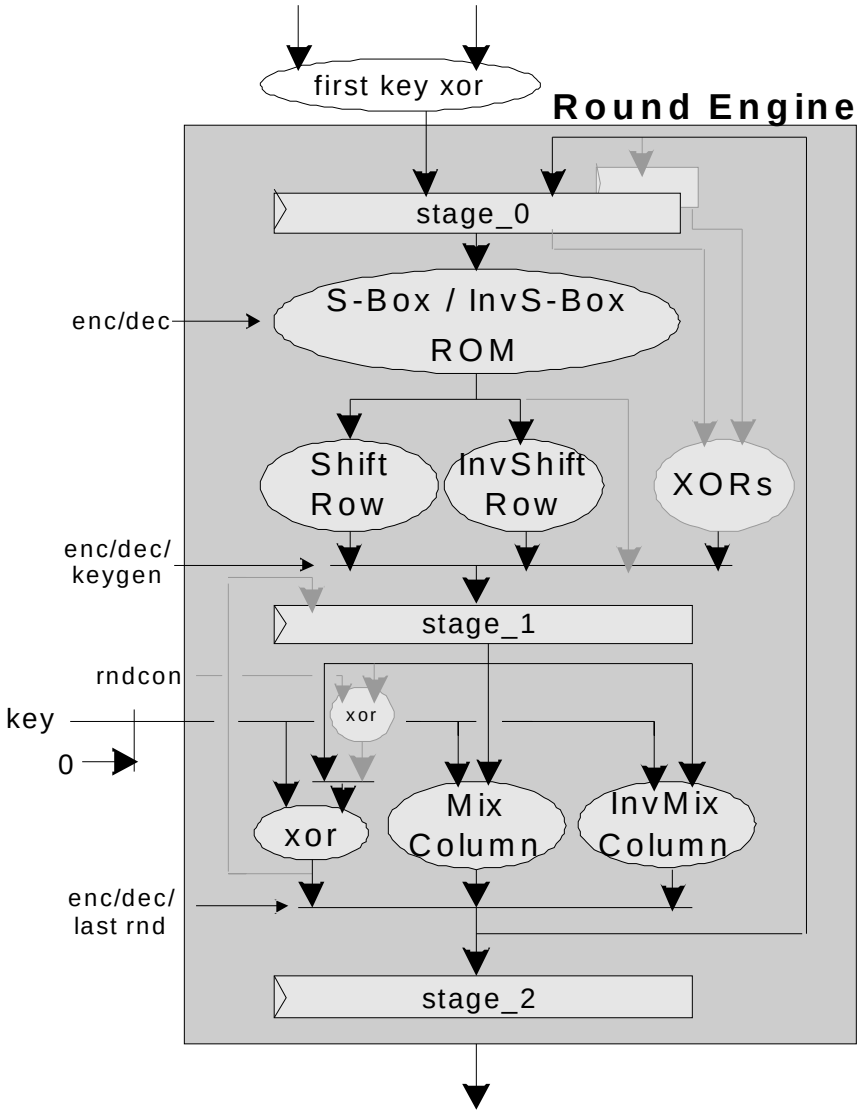
Microcode then loads the second data block into INPUT_1 register, and the first data block into INPUT_0 register (ECB mode always works on two data blocks). A load into INPUT_0 automatically starts the encryption operation. In the first pipeline stage, the first data block is brought from the INPUT_0 register, XOR'd with the first expanded key block (first key), and passed into the AES round engine. The collection of holding registers, muxes, and XORs are used by the various operating modes.

The AES round engine takes two clocks per round, but is pipelined to operate on two blocks at a time. Thus, on the clock after the first data block starts in the round engine, the second data block is XOR'd with the key and sent to the round engine. In our ECB example, the operation provides an effective throughput of one clock per round per 128-bit data block, and the results of each round feedback to the next round within the round engine. The appropriate extended key for each round is output from the key RAM to the round engine.

After all rounds are completed, the first-block result passes directly to the OUPUT_0 register for subsequent storing. The results for the second data block are passed to the OUTPUT_1 register.

Figure 4 illustrates the internal design of the AES round engine. This logic directly implements the AES algorithm. The lightly drawn lines represent additional logic (added to the base round logic) used to calculate the extended key sequence. Each round, for a single block, requires two pipeline stages.

Figure 4. Round Engine Hardware



Encryption is performed as shown in section 5.1 of FIPS-197 as summarized in Figure 5 of that document. The first stage performs the *SubBytes* S-Box substitution followed by the *ShiftRows* operation. The S-Box is implemented as sixteen 2x256x8-bit ROMs (2x for separate encryption and decryption values). The second pipeline stage performs the *MixColumns* operation and the XOR of the extended key. The key XOR is included as another leg in the sequence of XORs performing the mix column function and a bypass of the mix column function exists for the special last-round function.

Decryption is performed using the equivalent inverse cipher algorithm as described in section 5.3.5 and summarized in Figure 15 of FIPS-197. The first stage implements the *InvSubBytes* S-box substitution followed by the *InvShiftRows* operation. The second stage implements the *InvMixColumns* function. This path is the slowest timing path in the round engine due to the increased number of XOR levels over mix column.

The hardware occupies about 0.6 mm^2 and runs at the processor clock frequency.

2.2 SOFTWARE

All PadLock ACE functions are accessed using one x86 primary opcode. This is a group opcode providing eight sub-opcode functions. ACE instruction operands are defined by multiple memory values whose addresses are contained in General Purpose Registers. The approach is similar to that used by x86 string instruction formats.

The major components are:

- Two bits in the Centaur extended feature flags returned by the CPUID instruction. These bits identify (1) whether PadLock ACE physically exists on the processor and (2) whether it is currently enabled.
- A bit in the existing FCR MSR that allows a privileged program to enable or disable PadLock ACE
- One non-privileged x86 opcode that provides the PadLock ACE features. Figure 5 illustrates the instruction architecture and opcodes, and Figure 6 illustrates the eight group instructions.

This primary opcode is the same opcode for PadLock Random Number Generator introduced in the VIA Nehemiah processor (stepping 3). PadLock ACE defines five of the eight sub-opcodes. The remaining two sub-opcodes are **reserved** by VIA for additional PadLock features.

Figure 5. General PadLock ACE Architecture Format

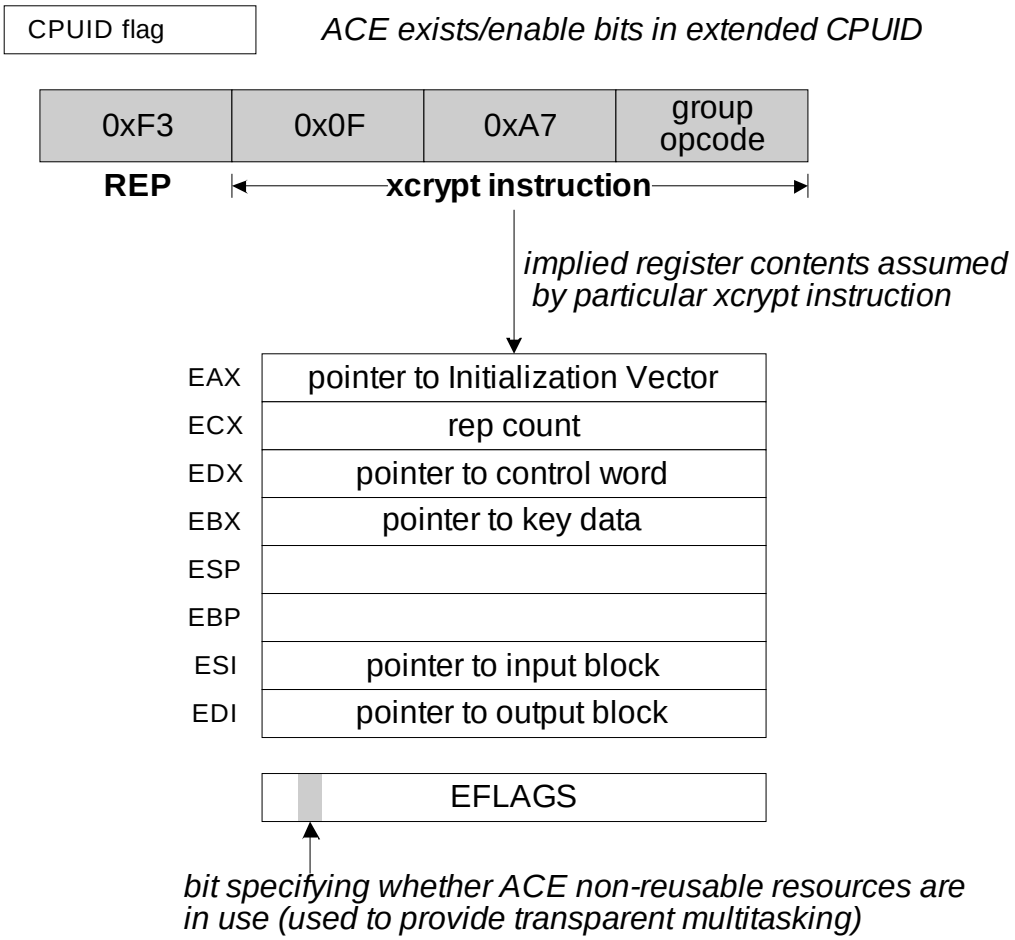


Figure 6. Instruction Functions

				instruction name	*group opcode*
0xF3	0x0F	0xA7	0xC0	**XSTORE (rng)**	**0**
0xF3	0x0F	0xA7	0xC8	**XCRYPTECB**	**1**
0xF3	0x0F	0xA7	0xD0	**XCRYPTCBC**	**2**
0xF3	0x0F	0xA7	0xD8	**XCRYPTCTR**	**3**
0xF3	0x0F	0xA7	0xE0	**XCRYPTCFB**	**4**
0xF3	0x0F	0xA7	0xE8	**XCRYPTOFB**	**5**
0xF3	0x0F	0xA7	0xF0	**invalid instruction**	**6**
0xF3	0x0F	0xA7	0xF8	**invalid instruction**	**7**

XCRYPT instructions are atomic with respect to the encryption of a single data block (two data blocks in ECB mode); that is, once execution starts, the specified function completes each block before interrupts are allowed.

XCRYPT instructions exist in REP form only. Like other REP x86 instructions, they can be interrupted and restarted. They differ from other REP instructions in that the performance of encrypting N blocks of data with a REP XCRYPT instruction is faster (as much as 50% faster) than the execution of N separate XCRYPT instructions.

XCRYPT instructions define their operands in x86 General Purpose Registers.

An EFLAGS register extension in Bit 30 allows multiple tasks to use XCRYPT instructions in a fashion transparent to applications and to the operating system.

The five supported modes are implemented by a collection of datapaths surrounding the basic AES round engine. The application program does not directly control these paths; instead, the application program merely specifies the operating mode to be used and the hardware and associated microcode manages the routing of data.

2.3 MULTI-TASKING

PadLock ACE is specifically designed to address two major (and related) goals:

- To allow multiple applications (including the operating system) to use XCRYPT without any operating systems support (such as a device driver), and

- To allow multiple applications (including the operating system) to use XCRYPT without any awareness of, or visibility of, other tasks (and their data) that are also using XCRYPT.

That is, if

- Task A executes an XCRYPT instruction, and
- A task switch subsequently occurs to task B that subsequently executes an XCRYPT instruction, the following must be true:

 Any inter-instruction state needed to later restart task A and execute subsequent task A XCRYPT instructions must be saved (when task A is suspended) and restored (when task A is restarted) by existing operating system code, and

 Any inter-instruction state required to execute task B's XCRYPT instructions correctly must be restored (when task B is started) by *existing* operating system code, and

 Task B must not be able to see, touch or smell anything (such as the keys) relating to task A's use of the XCRYPT instruction, and vice versa.

The ACE approach to satisfy these requirements is to make XCRYPT self-contained. That is, all of the information needed to perform an encryption is contained within the XCRYPT instruction by using pointers in general purpose registers to all of the operands. These pointers are saved across task switches by existing operating systems.

A problem with this approach is that loading the extended key sequence into the hardware registers takes a longer time than performing the encryption operation itself. To improve performance, ACE has a mechanism to avoid reloading keys into the hardware unless they have changed.

For all XCRYPT instructions, bit 30 in EFLAGS specifies whether the processor needs to load the extended key sequence and the control word from memory into the hardware registers:

- If EFLAGS:30 is 0 when an XCRYPT instruction is executed, the control word and extended key sequence are loaded into the hardware registers. EFLAGS:30 is then set to 1.
- If EFLAGS:30 is 1 when an XCRYPT instruction is executed, the control word and extended key sequence in the hardware registers are presumed correct ***and are not reloaded***. EFLAGS:30 is not changed.

In addition to this XCRYPT behavior, VIA processors implement changes to existing x86 instructions and operation that affect EFLAGS:

- EFLAGS:30 is set to 0 by *any* x86 instruction, interrupt, exception, task switch, etc. operation that causes EFLAGS to be stored (even if executed in 16-bit mode).
- EFLAGS:30 cannot be set to 1 by any x86 instruction that causes EFLAGS to be loaded. Only XCRYPT instructions set this bit to 1.
- EFLAGS:30 is set to 0 by a SYSENTER instruction.

The effect of this mechanism relative to multitasking is (starting at the beginning of time):

1. Task 1 executes a first XCRYPT instruction. Since EFLAGS:30 is 0, the XCRYPT logic loads the control word and extended keys into the hardware, and sets EFLAGS:30 to 1.
2. Task 1 executes a second XCRYPT instruction. Since EFLAGS:30 is 1, the XCRYPT logic bypasses loading the control word and extended keys. This improves the performance of the second XCRYPT instruction.

3. A task switch to task 2 occurs. The process of loading the context of task 2 always involves saving task 1's EFLAGS and loading task 2's EFLAGS. The save of task 1's EFLAGS sets bit 30 to 0 in the saved value. The load of task 2's EFLAGS may be done by the operating system with an explicit POPF instruction, or it may be performed by the built-in x86 exception or task switch mechanism. Regardless, this load causes task 2's EFLAGS:30 to be set to 0.

4. Task 2 executes an XCRYPT instruction. Since its EFLAGS:30 is 0, the XCRYPT logic loads the control word and extended keys into the hardware, and sets EFLAGS:30 to 1.

5. A task switch back to task 1 occurs. The EFLAGS:30 for task 1 has bit 30 cleared to 0, so the next task 1 XCRYPT execution will reload the task 1 control word and keys.

2.4 EXTENDED KEYS

An XCRYPT instruction always points to the encryption key. However, AES encryption requires an extended key sequence. This sequence provides a unique 128-bit value for each round of the algorithm. PadLock ACE supports two methods for providing these extended keys:

- **Hardware Generated.** The hardware can generate the extended key sequence for both encryption and decryption for 128-bit keys, for all specifiable round counts. This generation is performed as a side effect of an XCRYPT instruction that points to the base key and points to a control word that specifies hardware-generated extended keys. This option is the normal approach for 128-bit keys.

- **Application Provided.** The application can provide the entire extended key sequence instead of letting the hardware generate it. The load of this extended-key data is performed as a side effect of an XCRYPT instruction that points to the base key followed by the extended keys and points to a control word that specifies application-provided extended keys. This is the only supported option for 192-bit and 256-bit keys.

For hardware generated keys, the key data pointed to by EBX must start with byte 0 of the key (w[0] in AES notation) on a 16-byte aligned linear address, followed by byte 1 (w[1] in AES notation), up to byte 15.

For application-loaded keys for *encryption*, the extended key sequence starts with the normal key as described above followed immediately by the first byte of the extended key sequence, followed by the second byte, etc. for the entire sequence of extended keys. That is, the memory sequence w[0], w[1], etc. is as described in Figure 11 of FIPS-197.

For application-loaded keys for *decryption*, the *equivalent inverse cipher* extended keys must be provided in the inverse order (the real key comes last) as described in Figure 15 of FIPS-197. That is, the first addressed byte is defined in FIPS-197 Figure 15 as dw[0], the next is dw[1], and so forth.

For the application-loaded keys option, the hardware always loads sixteen 128-bit values from memory regardless of the specified key size. Any values beyond the normal extended key size are ignored and have no affect on the results, but that memory area must be accessible (within the segment limit, etc.)

NOTE: For calculation of *intermediate* results for *decryption*, for all key sizes, it is necessary for the application to load the extended key schedule.

2.5 XCRYPTECB: ELECTRONIC CODE BOOK (ECB) MODE

In this mode, each data block is encrypted independently from other blocks. Two blocks are pipelined through the round engine. One block comes from input_0 register, the other from input_1 register (via the hold register). Each data block is initially XOR'd with the first key block and the result feeds into the round engine. The result of each encryption round is feed back into the core for the next round. When all rounds have been completed, the final result is captured in one of the two output registers.

XCRYPTECB is supported on VIA Esther processors and VIA Nehemiah processors (stepping 8 and above)

Figure 7. XCRYPTECB Instruction

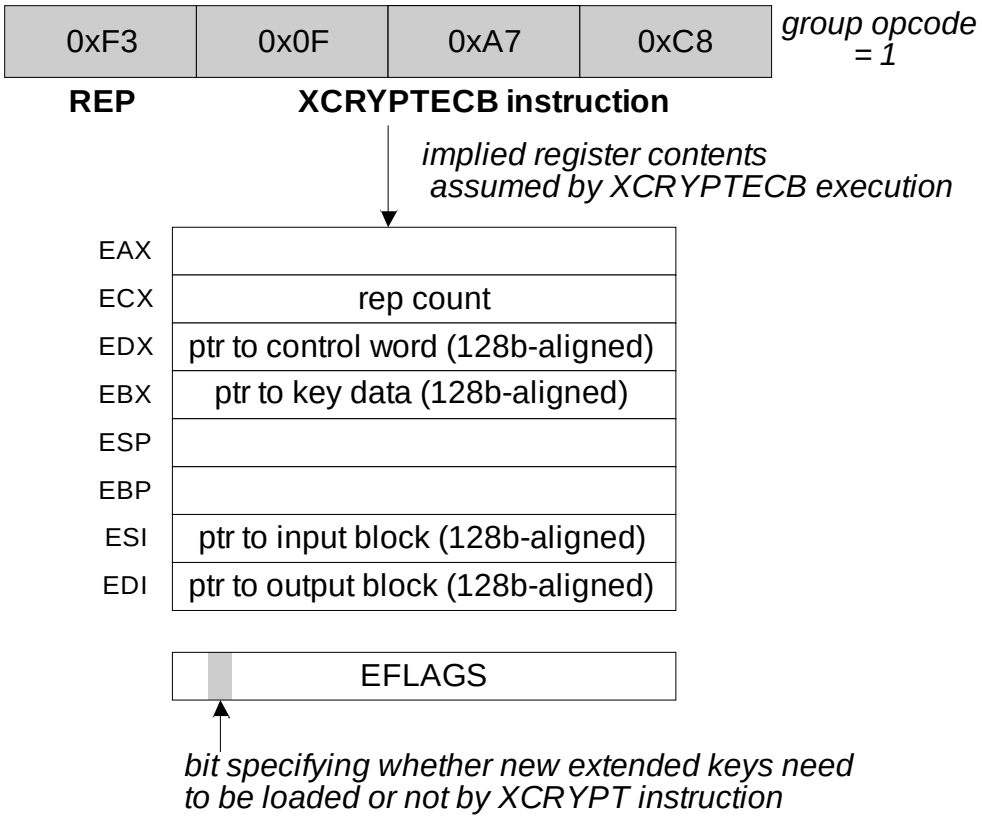


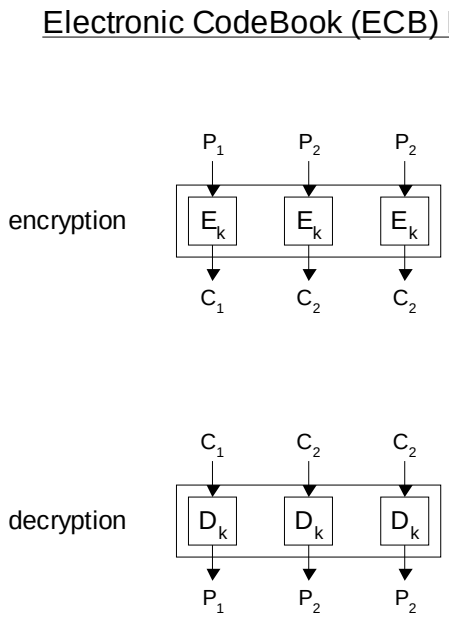
Figure 7 shows the detailed format of the REP XCRYPTECB instruction. This performs encryption using the Electronic Code Book (ECB) mode. The basic steps to use the instruction: **[Note: All addresses assume the ES segment selector, which may not be overridden]**

1. Check the Centaur extended CPUID flags to see if PadLock ACE is present.
2. If necessary, enable PadLock ACE using the FCR MSR. This is a privileged operation but rarely needs to be done, as the default state at RESET is that PadLock ACE is enabled.

3. Set bit 30 of EFLAGS to 0. This is interpreted by microcode to mean that the control word and extended keys provided by the subsequent REP XCRYPTECB instruction are new to this task and thus need to be loaded into internal hardware registers. This can be done by a PUSHF instruction followed by a POPF instruction. (Any load or store of EFLAGS zeroes bit 30). Be sure to do this whenever you use a new key.
4. Load the effective address (offset) of either (1) the entire extended key sequence, or (2) the key itself into EBX. A bit in the control word indicates the distinction between these two key-extension options.
5. Load the effective address (offset) of the desired ACE control word into EDX. The control word specifies encryption or decryption, the number of rounds, where the extended key comes from, and so forth.
6. Load the effective address (offset) of the first data block to be encrypted into ESI.
7. Load the effective address (offset) of the start of the ciphertext result area into EDI. **The source data address and the result data address may be identical; PadLock ACE supports encryption in place**.
8. Load the number of 128-bit data blocks to be encrypted into ECX. Note that each execution of the REP XCRYPTECB function processes two data blocks. When ECX is odd, one extra block will be encrypted but the result *will not be stored to memory*.
9. Execute the REP XCRYPTECB instruction.

Figure 8 illustrates the operation of ECB mode.

Figure 8. ECB Operating Mode Support



2.6 XCRYPTCBC: CIPHER BLOCK CHAINING (CBC) MODE

In this mode, the result of each encryption is XOR'd with the next incoming data block. This cipher block chaining (forwarding) operation insures that any single block of cipher text cannot be decrypted to plain text; only the entire file can be decrypted. This forwarding means that only one block is encrypted at a time; no pipelining of the round engine occurs. On the first round, an Initialization Vector (IV) provided by the application is used in place of the forwarded cipher text.

Unlike EBC, encryption and decryption have different routings through PadLock ACE.

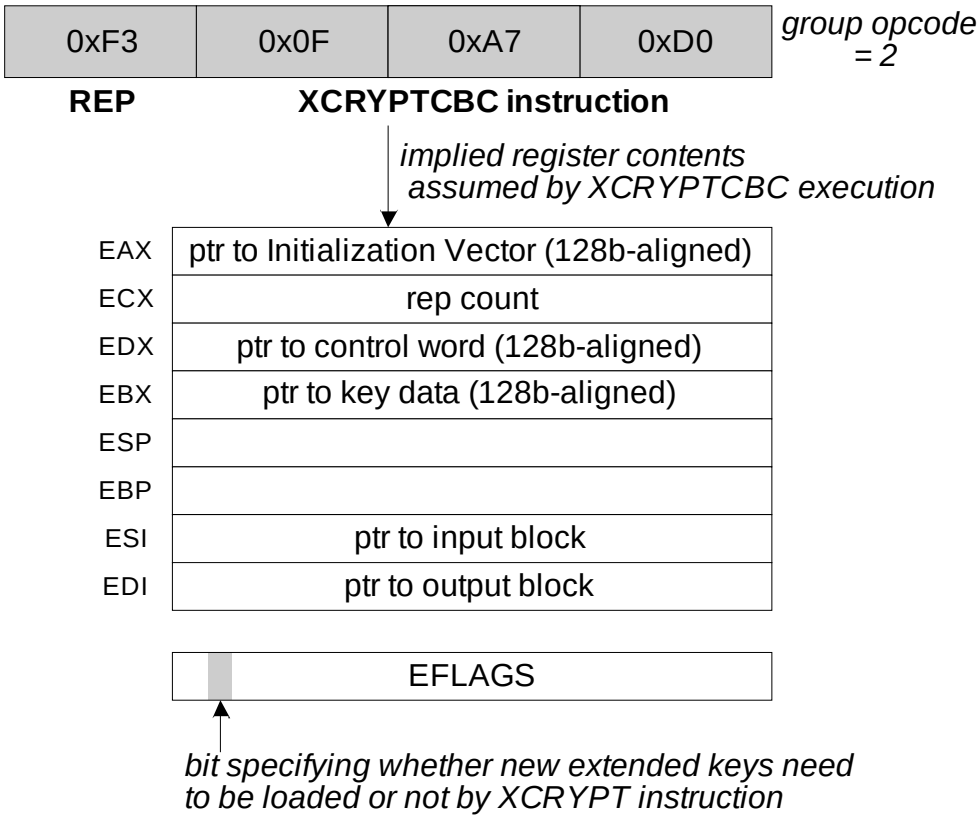
For CBC **encryption**, the plain text comes from the *input_0* register, is then XOR'd with either the IV or the forwarded cipher text results from the last block (from the *hold* register), and the results proceeds to the key XOR and the round engine. The encryption results are sent to the output register and the hold register for use on the following encryption.

For CBC **decryption**, the incoming cipher block is passed on to the key XOR and the round engine for decrypting. The incoming cipher block is also moved into the hold register for subsequent a XOR with the results of the round engine operating on the next subsequent cipher block.

XCRYPTCBC is supported on VIA Esther processors and VIA Nehemiah processors (stepping 8 and above)

Figure 9 shows the detailed format of the REP XCRYPTCBC instruction. This performs encryption using the Cipher Block Chaining (CBC) mode.

Figure 9. XCRYPTCBC instruction

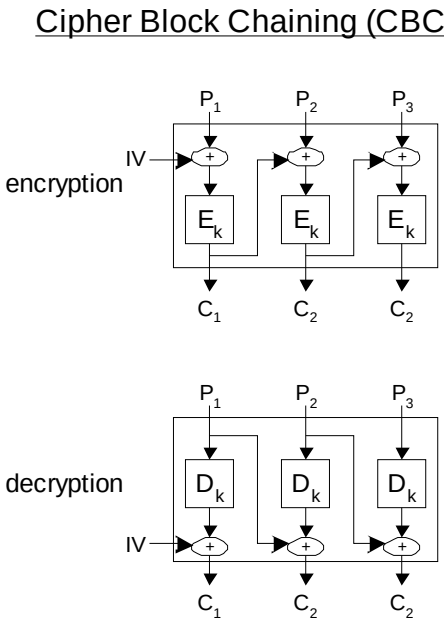


The basic steps to use the instruction: [**Note: All addresses assume the ES segment selector, which may not be overridden]**

10. Check the Centaur extended CPUID flags to see if PadLock ACE is present.
11. If necessary, enable PadLock ACE using the FCR MSR. This is a privileged operation but rarely needs to be done, as the default state at RESET is that PadLock ACE is enabled.
12. Set bit 30 of EFLAGS to 0. This is interpreted by microcode to mean that the control word and extended keys provided by the subsequent rep xcryptecb instruction are new to this task and thus need to be loaded into internal hardware registers. This can be done by a PUSHF instruction followed by a POPF instruction. (Any load or store of EFLAGS zeroes bit 30). Be sure to do this whenever you use a new key.
13. Load the effective address (offset) of either (1) the entire extended key sequence, or (2) the key itself into EBX. A bit in the control word indicates the distinction between these two key-extension options.
14. Load the effective address (offset) of the desired ACE control word into EDX. The control word specifies encryption or decryption, the number of rounds, where the extended key comes from, whether the plaintext and ciphertext buffers are known to be aligned on 16-byte boundaries, and so forth.
15. Load the effective address (offset) of the first data block to be encrypted into ESI.
16. Load the effective address (offset) of the Initialization Vector into EAX.
17. Load the effective address (offset) of the start of the ciphertext result area into EDI. **The source data address and the result data address may be identical; PadLock ACE supports encryption in place**.
18. Load the number of 128-bit data blocks to be encrypted into ECX.
19. Execute the REP XCRYPTCBC instruction.

Figure 10 illustrates the operation of CBC mode.

Figure 10. CBC Operating Mode Support

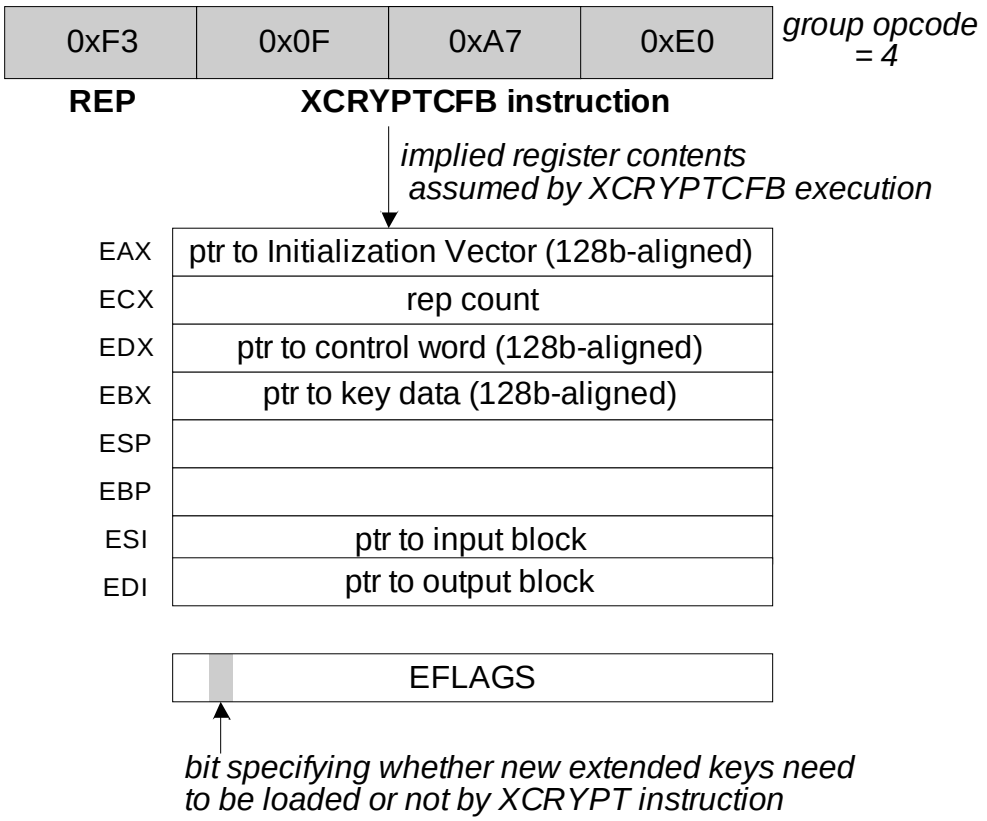


2.7 XCRYPTCFB: CIPHER FEEDBACK (CFB) MODE

In this mode, the Initialization Vector (IV) is encrypted and the result is XOR'd with the incoming plaintext (ciphertext) block. This is the ciphertext (plaintext) result, which is forwarded to the subsequent round as the IV for that round. This block chaining (forwarding) operation insures that any single block of cipher text cannot be decrypted to plain text; only the entire file can be decrypted. This forwarding means that only one block is encrypted at a time; no pipelining of the round engine occurs. On the first round, an Initialization Vector (IV) provided by the application is used in place of the forwarded cipher text.

XCRYPTCFB is supported on VIA Esther processors and VIA Nehemiah processors (stepping 8 and above) Figure 11 shows the detailed format of the REP XCRYPTCFB instruction. This performs encryption using the Cipher FeedBack (CFB) mode.

Figure 11. XCRYPTCFB instruction



The basic steps to use the instruction: [**Note: All addresses assume the ES segment selector, which may not be overridden]**

20. Check the Centaur extended CPUID flags to see if PadLock ACE is present.

21. If necessary, enable PadLock ACE using the FCR MSR. This is a privileged operation but rarely needs to be done, as the default state at RESET is that PadLock ACE is enabled.

22. Set bit 30 of EFLAGS to 0. This is interpreted by microcode to mean that the control word and extended keys provided by the subsequent rep xcryptecb instruction are new to this task and thus need to be loaded into internal hardware registers. This can be done by a PUSHF instruction followed by a POPF instruction. (Any load or store of EFLAGS zeroes bit 30). Be sure to do this whenever you use a new key.

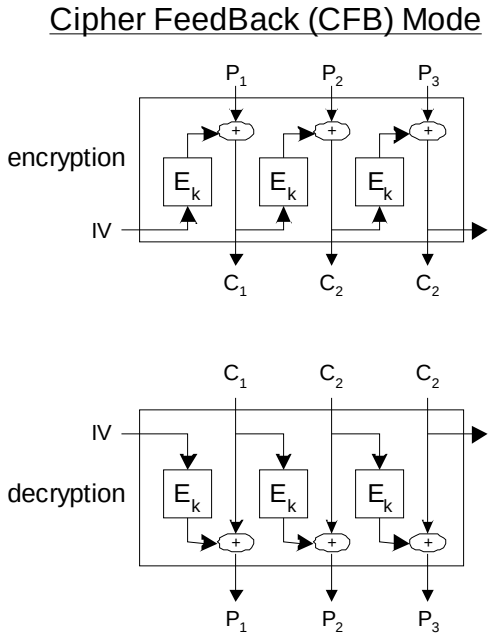
23. Load the effective address (offset) of either (1) the entire extended key sequence, or (2) the key itself into EBX. A bit in the control word indicates the distinction between these two key-extension options.

24. Load the effective address (offset) of the desired ACE control word into EDX. The control word specifies encryption or decryption, the number of rounds, where the extended key comes from, whether the plaintext and ciphertext buffers are known to be aligned on 16-byte boundaries, and so forth.

25. Load the effective address (offset) of the first data block to be encrypted into ESI.
26. Load the effective address (offset) of the Initialization Vector into EAX.
27. Load the effective address (offset) of the start of the ciphertext result area into EDI. **The source data address and the result data address may be identical; PadLock ACE supports encryption in place**.
28. Load the number of 128-bit data blocks to be encrypted into ECX.
29. Execute the REP XCRYPTCFB instruction.

Figure 12 illustrates the operation of CFB mode.

Figure 12. CFB Operating Mode Support



2.8 XCRYPTOFB: OUTPUT FEEDBACK (OFB) MODE

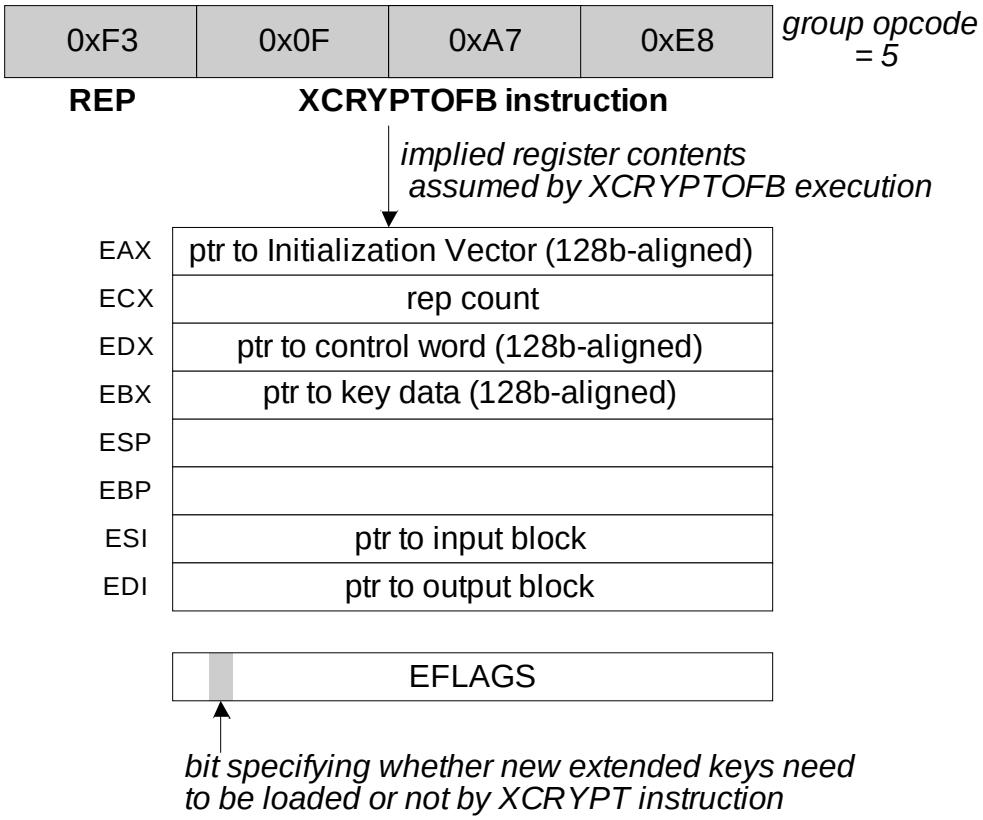
In this mode, the Initialization Vector is encrypted and the result is forwarded to the next round for further encryption, yielding a sequence of 128-bit blocks. This sequence is XOR'd with the plaintext to produce the ciphertext, or with the ciphertext to reconstruct the plaintext.

OFB mode has often been used to pre-calculate a byte stream to XOR with a message as the encryption of the Initialization Vector and the XOR with the plaintext can be separated in time. Thus OFB – when used correctly – is like a "one time pad". It is thus critical that no 128-bit block of the encrypted sequence be re-used.

XCRYPTOFB is supported on VIA Esther processors and VIA Nehemiah processors (stepping 8 and above)

Figure 13 shows the detailed format of the REP XCRYPTOFB instruction. This performs encryption using the Output FeedBack (OFB) mode.

Figure 13. XCRYPTOFB instruction



The basic steps to use the instruction: [**Note: All addresses assume the ES segment selector, which may not be overridden]**

1. Check the Centaur extended CPUID flags to see if PadLock ACE is present.

2. If necessary, enable PadLock ACE using the FCR MSR. This is a privileged operation but rarely needs to be done, as the default state at RESET is that PadLock ACE is enabled.

3. Set bit 30 of EFLAGS to 0. This is interpreted by microcode to mean that the control word and extended keys provided by the subsequent rep xcryptecb instruction are new to this task and thus need to be loaded into internal hardware registers. This can be done by a PUSHF instruction followed by a POPF instruction. (Any load or store of EFLAGS zeroes bit 30). Be sure to do this whenever you use a new key.

4. Load the effective address (offset) of either (1) the entire extended key sequence, or (2) the key itself into EBX. A bit in the control word indicates the distinction between these two key-extension options.

5. Load the effective address (offset) of the desired ACE control word into EDX. The control word specifies encryption or decryption, the number of rounds, where the extended key comes from, whether the plaintext and ciphertext buffers are known to be aligned on 16-byte boundaries, and so forth.

6. Load the effective address (offset) of the first data block to be encrypted into ESI.

7. Load the effective address (offset) of the Initialization Vector into EAX.

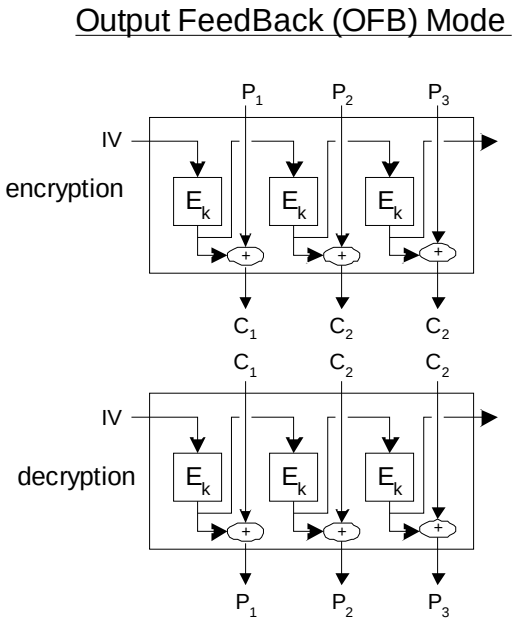
8. Load the effective address (offset) of the start of the ciphertext result area into EDI. **The source data address and the result data address may be identical; PadLock ACE supports encryption in place**.

9. Load the number of 128-bit data blocks to be encrypted into ECX.

10. Execute the REP XCRYPTOFB instruction.

Figure 14 illustrates the operation of OFB mode.

Figure 14. OFB Operating Mode Support



2.9 XCRYPTCTR: COUNTER (CTR) MODE

In this mode, and like OFB mode, the Initialization Vector is encrypted, and the result is XOR'd with the plaintext to produce the ciphertext, or with the ciphertext to reconstruct the plaintext. However, no data is forwarded to the next round.

Like OFB mode, CTR mode can be thought of as generating a “one-time-pad”. It is critical that no 128-bit block of the encrypted sequence be re-used. In CTR mode, this is accomplished by incrementing the CTR, or more accurately a specific bit-field of the CTR

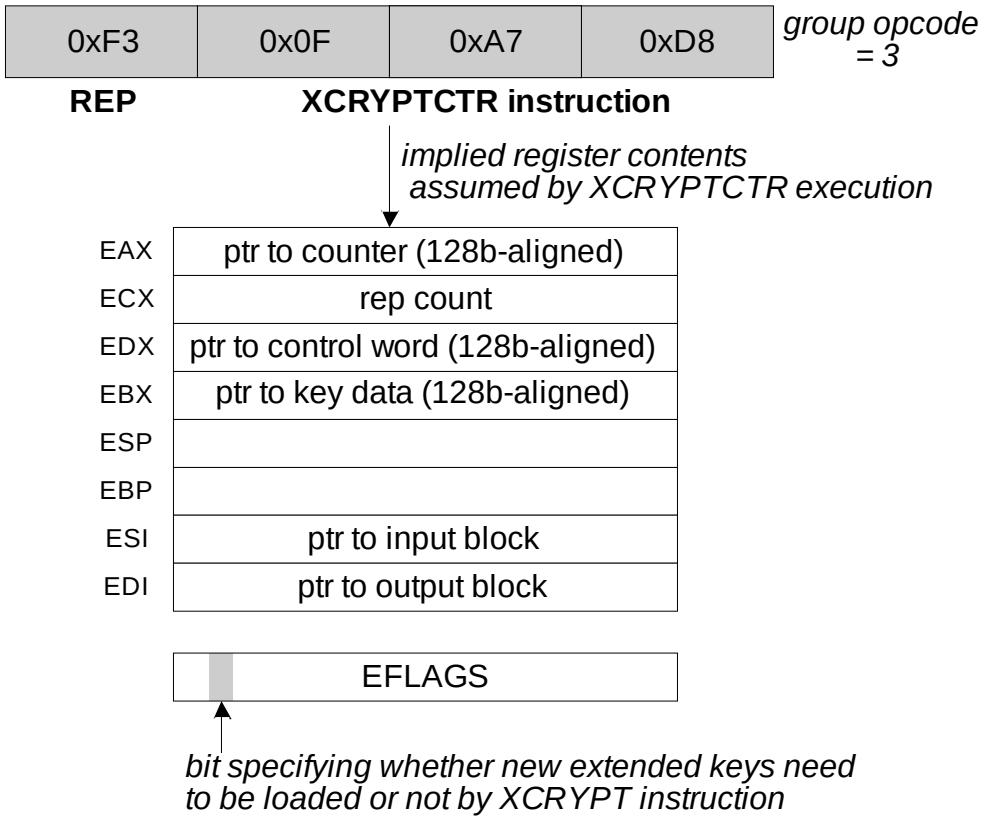
There are many possible counter modes. PadLock ACE implements a counter where the first 14 bytes of the counter (in normal ascending address order) are the **nonce,** a random value specific to the encryption of a particular message and which should never be re-used. The last two bytes are considered to be the 16-bit integer counter ***in big-endian format***.

The counter mode is compatible with the CCM mode as specified by wireless protocol 802.11i and was chosen specifically by Centaur Technology and VIA to be compatible with that protocol.

XCRYPTCTR is supported on the VIA Esther processor.

Figure 15 shows the detailed format of the REP XCRYPTCTR instruction. This performs encryption using the Counter (CTR) mode.

Figure 15. XCRYPTCTR instruction



The basic steps to use the instruction: [**Note: All addresses assume the ES segment selector, which may not be overridden]**

11. Check the Centaur extended CPUID flags to see if PadLock ACE is present.

12. If necessary, enable PadLock ACE using the FCR MSR. This is a privileged operation but rarely needs to be done, as the default state at RESET is that PadLock ACE is enabled.

13. Set bit 30 of EFLAGS to 0. This is interpreted by microcode to mean that the control word and extended keys provided by the subsequent rep xcryptecb instruction are new to this task and thus need to be loaded into internal hardware registers. This can be done by a PUSHF instruction followed by a POPF instruction. (Any load or store of EFLAGS zeroes bit 30). Be sure to do this whenever you use a new key.

14. Load the effective address (offset) of either (1) the entire extended key sequence, or (2) the key itself into EBX. A bit in the control word indicates the distinction between these two key options.

15. Load the effective address (offset) of the desired ACE control word into EDX. The control word specifies the number of rounds, where the extended key comes from, and so forth.

16. Load the effective address (offset) of the first data block to be encrypted into ESI.

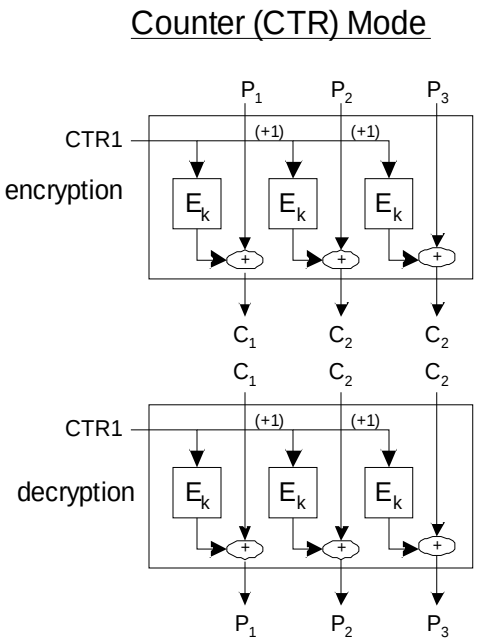
17. Load the effective address (offset) of the Counter into EAX.

18. Load the effective address (offset) of the start of the ciphertext result area into EDI. **The source data address and the result data address may be identical; PadLock ACE supports encryption in place**.

19. Load the number of 128-bit data blocks to be encrypted into ECX.
20. Execute the REP XCRYPTCTR instruction.

Figure 16 illustrates the operation of CTR mode.

Figure 16. CTR operating mode support



CHAPTER

3

PROGRAMMING DETAILS

3.1 MEMORY FORMATS

Most PadLock ACE data loads and stores must be 16-byte aligned. Relative to AES notation, byte 0 is at the lowest address, byte 1 is at the next higher address, and so forth. That is, data is organized in the normal x86 little endian fashion such that AES byte 0 is the normal x86 architecture byte 0 (the linear address provided by an instruction operand).

The VIA Nehemiah ACE required that *all* ACE loads and stores be 16-byte aligned. The VIA Esther processor has added a new algorithm (defined in bit 5 of the control word – see next section) that performs AES on plaintext and ciphertext with arbitrary alignment. This new algorithm is supported by all AES operating modes ***except ECB***.

Performance of the new algorithm will still of course be best if the plaintext and ciphertext are nevertheless aligned on a 16-byte boundary. However, when the plaintext and ciphertext buffers are known to be aligned on a 16-byte boundary, the original aligned-only algorithm is preferred, as the performance is significantly better for CBC, CFB, and OFB modes. On future VIA processors the performance difference may be reduced, but in all cases (where the plaintext and ciphertext buffers are aligned on 16-byte boundaries) the aligned-only algorithm will be faster.

3.2 CONTROL WORD POINTER

EDX (or DX) in all XCRYPT instructions contains the effective address of the ACE control word. This address is always relative to segment ES. The resulting linear address must be aligned on a 128-bit boundary, otherwise a General Protection exception occurs.

Whether DX or EDX is used is based on the effective address size for the executed instruction.

When the new 16-byte alignment control bit is set (bit 5 = 1), XCRYPT instructions require that 112 bytes of scratch memory be allocated immediately after the control word. (A total of 128 bytes at ES:EDX) This memory is used by the microcode of XCRYPT instructions to move unaligned plaintext and ciphertext data to 16-byte aligned buffers when required by PadLock ACE hardware.

The control word fields are:

127:12	11:10	9:9	8:8	7:7	6:6	5:5	4:4	3:0
Reserved 0	Key size	Decrypt/ encrypt	Intermediate/ normal	Key generation	Algorithm	16-byte alignment	Digest Mode	Round count

Bits 3:0 Round count: This four-bit field contains the number of rounds to be used for the encryption or decryption operation. Although the number of rounds for each key size of AES is specified by the FIPS standard, this value is not built into the hardware; the application program must specify the number of rounds**.** *Note that a round count of 0 will encrypt or decrypt for 16 rounds.* Assembly language programmers can think of the round count as a "loop" index, not a "rep" counter, in a 4-bit register.

Bits 4:4 Digest mode: This one-bit field specifies whether the CBC and CFB modes should operate normally (bit 4 = 0), when each encrypted plaintext block (XOR'd per operating mode) is stored in a separate ciphertext block. In Digest mode (bit 4 = 1), the ciphertext buffer is not incremented and the AES calculates what is known as a Message Authentication Code.

This mode is supported only on the VIA Esther processor. On VIA Nehemiah processors a non-zero value in this field is undefined.

Message Authentication Code is defined by the standards only for encryption. However, PadLock ACE hardware will not increment the pointer specified in ES:EDI for the CBC and CFB operating modes regardless of the state of the encrypt/decrypt bit of the control word.

Bits 5:5 16-byte alignment: This one-bit field specifies whether the plaintext and ciphertext buffers must be 16-byte aligned. When bit 5 = 0, which was required for VIA Nehemiah ACE, the plaintext and ciphertext buffers must be 16-byte aligned. When bit 5 = 1, this requirement is relaxed, and the VIA Esther ACE executes microcode that internally copies the non-aligned plaintext and ciphertext buffers to and from known aligned memory so that the hardware (which still does require 16-byte alignment) will operate correctly.

*This mode is supported only on the VIA Esther processor***.** On VIA Nehemiah processors a non-zero value in this field is undefined

The 16-byte alignment for plaintext and ciphertext is still required for ECB mode, and XCRYPTECB will fault if bit 5 = 1 and either plaintext or ciphertext is not 16-byte aligned.

Bits 6:6 Algorithm: This one-bit field specifies the cryptographic algorithm. The only algorithm currently supported is the AES, which has a field value of 0. Other values are reserved and PadLock ACE behavior is undefined if this field is set to a reserved value.

The VIA Nehemiah processor supported only one algorithm in PadLock ACE: the AES with the plaintext and ciphertext buffers mandatory aligned on 16-byte boundaries. As non-zero values for bits 4 and 5 were undefined for the VIA Nehemiah ACE, existing software written for that processor will work as before and will execute the same microcode sequences optimized for (and assuming) 16-byte alignment of the plaintext and ciphertext buffers.

Bits 7:7 Key Generation: This one-bit field specifies whether the extended keys are to be generated by the hardware (0) or are to be loaded from memory (1). Hardware key generation is supported only for 128-bit keys, and setting bit 7 = 0 when bit 10 = 1 or bit 11 = 1 (setting key size to other than 128 bits) will result in undefined behavior by PadLock ACE.

Bits 8:8 Intermediate/Normal: This one-bit field specifies normal operation (0) or intermediate operation (1). Intermediate mode allows examination of the result of intermediate rounds.

Bits 9:9 Encrypt/Decrypt: This one-bit field specifies encryption (0) or decryption (1).

Bits 11:10 Key size: This two-bit field specifies the key size: 128-bit (0), 192-bit (1), 256-bit (2), and reserved (3). PadLock ACE behavior is undefined if key size is set to the reserved value.

Bits 127:12 Reserved: These bits should be set to 0 and the contents of this field should not be used by the application.

3.3 DATA REGISTERS

INPUT DATA POINTER: ESI

ESI (or SI) in all XCRYPT instructions contains the effective address (offset) of the input data. This address is always relative to the segment specified in ES. If Control Word bit 5 = 0, the resulting linear address must be aligned on a 128-bit boundary, otherwise a General Protection exception occurs. If Control Word bit 5 = 1 this alignment requirement is relaxed *except* for ECB mode.

Non 16-byte aligned data (Control Word bit 5 = 1) is not supported on VIA Nehemiah processors.

For *encryption* operations, ESI (or SI) points to the **plaintext**. For *decryption* operations, ESI (or SI) points to the **ciphertext**.

At the end of instruction execution, or if an interrupt (or page fault, etc.) has occurred, ESI (or SI) is incremented by the number of bytes *stored*. The address is always incremented: EFLAGS.DF has no effect. Whether SI or ESI is used and updated is based on the effective address size for the executed instruction.

CIPHERTEXT POINTER: EDI

EDI (or DI) in all XCRYPT instructions contains the effective address (offset) of the result data generated by the instruction. This address is always relative to the segment specified in ES. If the Control Word bit 5 = 0, the resulting linear address must be aligned on a 128-bit boundary, otherwise a General Protection exception occurs. If Control Word bit 5 = 1 this alignment requirement is relaxed *except* for ECB mode.

Non 16-byte aligned data (Control Word bit 5 = 1) is not supported on VIA Nehemiah processors.

For *encryption* operations, EDI (or DI) points to the **ciphertext**. For *decryption* operations, EDI (or DI) points to the **plaintext.**

At the end of instruction execution, or if an interrupt (or page fault, etc.) has occurred, EDI (or DI) is incremented by the number of bytes *stored*. The address is always incremented: EFLAGS.DF has no effect. Whether DI or EDI is used and updated is based on the effective address size for the executed instruction.

> **EXCEPTION**: when Control Word bit 4 = 1, and for CBC and CFB operating modes only, the Ciphertext Pointer in EDI (or DI) is never incremented.

ENCRYPTION KEY POINTER: EBX

EBX (or BX) in all XCRYPT instructions contains the effective address (offset) of the key data. This address is always relative to the segment specified in ES. The resulting linear address must be aligned on a 128-bit boundary, otherwise a General Protection exception occurs. Whether BX or EBX is used is based on the effective address size for the executed instruction.

The key data takes one of two forms depending on bit 7 of the control word:

- Bit 7 = 0: The primary key. The extended key sequence will be generated by the hardware.
- Bit 7 = 1: The primary key followed by the application calculated extended key sequence.

NUMBER OF BLOCKS TO ENCRYPT: ECX

ECX (or CX) in all REP XCRYPT instructions contains the count of 128-bit data blocks to be processed.

At the end of instruction execution, ECX (or CX) will be zero. If an interrupt (or page fault, etc.) has occurred, CX or ECX will be decremented by the number of blocks processed and stored by the instruction prior to the interrupt. Whether CX or ECX is used and updated is based on the effective address size for the executed instruction.

INITIALIZATION VECTOR POINTER: EAX

AX or EAX in chaining mode XCRYPT instructions (all modes except ECB) contains the effective address (offset) of the initialization vector (IV). In Counter (CTR) mode, the Initialization Vector is more accurately called the *counter*. This address is always relative to the segment specified in ES. The resulting linear address must be aligned on a 128-bit boundary, otherwise a General Protection exception occurs. The choice of whether AX or EAX is used is based on the effective address size for the executed instruction.

As for normal x86 string instructions, if an external interrupt occurs during the processing of the REP XCRYPT instruction, the registers are updated before the interrupt transition occurs such that the REP instruction may be restarted upon return from interrupt handling. The initialization vector presents a problem since the original initialization vector is not the right one to use upon instruction restart. So the XCRYPT logic changes either the address in AX/EAX, or changes the data value pointed to by AX/EAX to be the appropriate initialization vector for the subsequent iteration.

Thus, an application program must not assume that EAX is unchanged, or that the value pointed to by EAX is unchanged, after the execution of a chaining mode REP XCRYPT instruction. For OFB mode, and for ***decryption*** *using CBC and CFB modes, the value pointed to by EAX will be changed. For* ***encryption*** *using CBC and CFB modes, the value in EAX itself is changed*

For CTR mode, the high-order 16 bits of the value pointed to by EAX will be incremented by one for every block processed. ***Note also that this 16-bit counter is big-endian, compatible with the counter mode defined for Wireless specification 801.11i***

3.4 REP PREFIX

The REP XCRYPT instructions have similarities with the x86 REP MOVS instruction:

- The value in EDI is updated with the number of result bytes stored (except for CBC and CFB modes when the Control Word Digest Mode bit is set).
- An address-size prefix affects the size of EAX, EBX, EDX, ESI, and EDI used
- The usual address exceptions (past the segment limit, page fault, etc.) can occur

And there are some different semantics:

- The value in ESI is updated with the number of result bytes stored. Note that in ECB mode with an initial odd value in ECX, one more data block from ESI is processed than is stored.
- A REPNE prefix causes undefined behavior.
- An operand size prefix causes an Invalid Instruction exception.
- The DF (direction flag) in EFLAGS has no effect. The operation always proceeds from low address to high address.

3.5 EXCEPTION HANDLING

During execution of an XCRYPT instruction, any number of exceptions or interrupts may occur.

For most exceptions, microcode instructions appear, logically, just like atomic x86 instructions: that is, the exception or interrupt occurs only before the instruction begins, or after it ends.

For normal x86 REP instructions, these exceptions occur “naturally” when the base instruction being repeated has completed, and after all architecture registers such as source and data pointers, and the REP counter have been updated. PadLock ACE XCRYPT instructions behave the same way with respect to these controlled exceptions and interrupts.

However some exceptions occur “in the middle of” the microcode sequence. A good example is a page fault that occurs when loading a plaintext block from ES:[ESI] into the input_0 register. From the perspective of the XCRYPT instruction, the page fault occurs “instantaneously”. Execution branches to some fault handler, and eventually (well, almost always) the original XCRYPT instruction will be re-issued by the translator.

Thus the microcode must not change an architecturally visible register until it is guaranteed that no page fault or similar exception can occur, which would cause an incorrect state on instruction restart. This is handled by not changing the x86 registers until after an ACE store instruction has been issued and is waiting in the SSE queue. On VIA processors, that store instruction is guaranteed to complete, and so only then are the x86 visible registers updated to reflect the state of the AES operation at the completion of the store.

Normal x86 REP instructions, when ECX == 0, are null operations. However, REP XCRYPTECB instructions with ECX == 0 can page fault. This is because the microcode does not check for a null REP count until after the first data blocks have been loaded, and those load operations could page fault. The advantage of starting the AES round engine immediately is that a single encryption operation runs some 3% to 4% faster than if ECX were tested first, because the AES round engine executes in parallel with the integer pipeline that executes the microcode.

CHAPTER

4

USING PADLOCK ACE

4.1 ENABLING

To use VIA PadLock ACE, three conditions must be met:

- PadLock ACE must physically exist on the chip. Its existence can be discovered from the Centaur Extended CPUID Feature Flags. This condition is permanently set as part of the manufacturing process.
- PadLock ACE must be enabled. The enabled state can be also discovered from the Centaur Extended CPUID Feature Flags. PadLock ACE can be enabled/disabled by writing to the Function Control Register (FCR), MSR 0x1107. The RESET default is enabled. But, see the next condition.
- SSE instructions must be enabled via the standard x86 method of enabling the FXSAVE/FXRSTOR instructions using CR4[9] This CR4 enabling also enables the full set of SSE instructions on Intel processors and on VIA Esther and Nehemiah processors.

VIA Esther and Nehemiah processors support the Centaur Extended Feature Flags. When the CPUID instruction is executed with EAX=0xC0000000, the processor returns 0xC0000001 in EAX. Note that earlier VIA C3 processors do not support the Centaur Extended CPUID Functions and will return EAX=0xC0000000 or EAX=0x00000000 (see the appropriate VIA processor datasheet for details on the extended CPUID function).

VIA Esther processors implement additional cryptographic functions not supported by VIA Nehemiah processors. These include the Secure Hash Algorithm and Montgomery Multiplier capabilities described in the introduction, and together they are known as PadLock ACE2. If you have a Nehemiah processor rather than an Esther processor, the extended CPUID bits are bits 6 and 7 (which describe PadLock ACE) rather than bits 8 and 9 (which describe PadLock ACE2). Note that on the VIA Esther processor, CPUID bits 6 and 7 will be copies of CPUID bits 8 and 9, respectively.

If a CPUID with EAX=0xC0000000 returns a value in EAX >= 0xC0000001 then Centaur Extended Feature Flags are supported. A CPUID with EAX=0xC0000001 then returns the Centaur Extended Feature Flags in EDX. There are four bits in EDX that describe PadLock ACE and PadLock ACE2

- EDX[6] == 0 PadLock ACE does not exist on this chip. FCR[28] cannot be set to enable PadLock ACE and the XCRYPT instructions cause an Invalid Opcode Fault.

- EDX[6] == 1 PadLock ACE exists. The behavior of XCRYPT instructions is dependent upon whether or not PadLock ACE is enabled.
- EDX[7] == 0 PadLock ACE is disabled. XCRYPT instructions cause an Invalid Opcode Fault. If PadLock ACE is present, FCR[28] can be set to enable PadLock ACE
- EDX[7] == 1 PadLock ACE is enabled (usually the RESET default. XCRYPT instructions behave as defined in this application note (provided CR4[9] =1)
- EDX[8] == 0 PadLock ACE2 does not exist on this chip. FCR[28] cannot be set to enable PadLock ACE2 and the XCRYPT instructions cause an Invalid Opcode Fault.
- EDX[8] == 1 PadLock ACE2 exists. The behavior of XCRYPT instructions is dependent upon whether or not PadLock ACE2 is enabled.
- EDX[9] == 0 PadLock ACE2 is disabled. XCRYPT instructions cause an Invalid Opcode Fault. If PadLock ACE2 is present, FCR[28] can be set to enable PadLock ACE2
- EDX[9] == 1 PadLock ACE2 is enabled (usually the RESET default. XCRYPT instructions behave as defined in this application note (provided CR4[9] =1)

VIA Esther and Nehemiah processors implement a Function Control Register (FCR), MSR 0x1107, which can be written by software to enable or disable various features. One bit controls PadLock ACE and PadLock ACE2 - FCR[28]:

- FCR[28] = 0 PadLock ACE and PadLock ACE2 (for VIA Esther processors) are not enabled. If EDX[8] indicates that PadLock ACE is present, a WRMSR can set this bit to 1 thus enabling PadLock ACE.
- FCR[28] = 1 PadLock ACE2 and PadLock ACE2 (for VIA Esther processors) are enabled. A WRMSR can set this bit to 0 thus disabling the PadLock ACE.

4.2 STARTUP

The RESET signal is an immediate asynchronous process: RESET halts operation immediately regardless of the state of the processor. As part of the RESET process, the following PadLock ACE actions are performed:

- Any in-progress XCRYPT instruction is immediately cancelled and undefined results may be stored.
- The FCR is reset to its default value. This may or may not directly change PadLock ACE enable status.
- The CR4 register is reset to zero. Since this resets the FXSAVE/FXRSTOR enable, (CR4[9]) PadLock ACE is indirectly disabled.
- EFLAGS:30 is cleared to zero.

The INIT signal is an interrupt that occurs between x86 instructions: INIT never halts instructions in the middle of their execution (except for REP string instructions, where INIT behaves like INTR). The effect of INIT on PadLock ACE is:

- The CR4 register is reset to zero. Since this resets the FXSAVE/FXRSTOR enable, (CR4[9]) PadLock ACE is indirectly disabled.
- EFLAGS:30 is cleared to zero.

4.3 MANUFACTURED STATE

PadLock ACE is tested as part of the VIA processor manufacturing process. Relative to potential fault coverage, however, this level of testing is not 100% complete. For certain sensitive applications, additional testing (by the application) may be desirable.

4.4 VERIFICATION

Cryptography Research, Inc., of San Francisco has performed an independent evaluation of the AES implementation of the ACE unit as implemented in stepping 8 of the VIA Nehemiah processor. A copy of their (preliminary) report is available on the VIA web site:

http://www.via.com.tw/en/images/Products/eden/pdf/CRI%20Evaluation%20of%20PadLock%20ACE.pdf

It is also posted at Cryptography Research:

http://www.cryptography.com/resources/whitepapers/Centaur_AESstatement.pdf

CHAPTER

5

INSTRUCTION TIMING

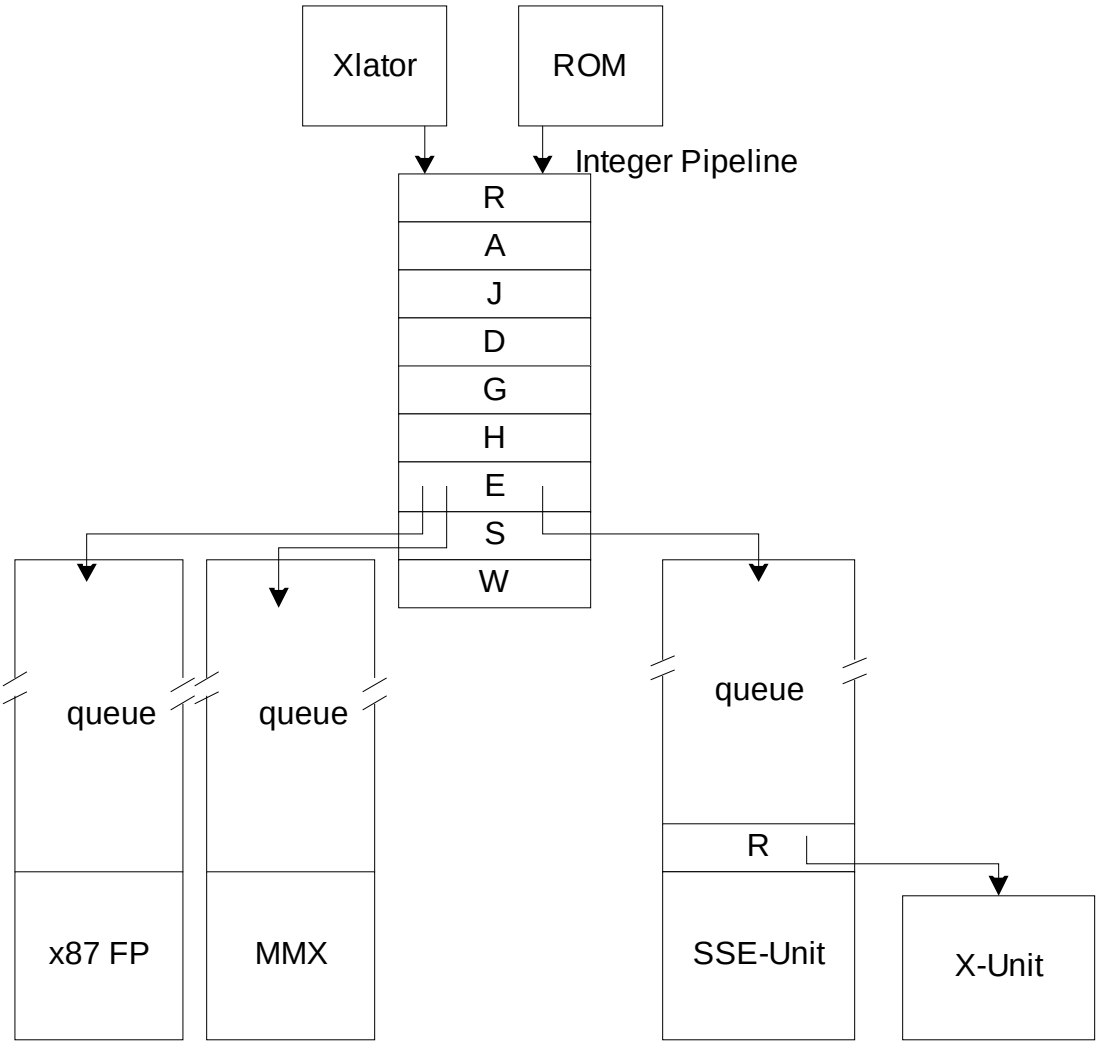
5.1 TIMING BASICS

As for all x86 instructions that reference memory, the timing of XCRYPT instructions is non-deterministic due to cache misses, page faults, interrupts, load and store buffer stalls, SSE queue-full stalls, external DMA snoops, etc. Consistent with other published x86 instruction timings, this section ignores these variables and assumes a perfect environment: everything is in the cache, no interrupts, no snoops, no stalls, etc.

ACE instruction timing is more complicated in that a large number of XCRYPT execution clocks are overlapped with subsequent instruction execution clocks. To understand this, Figure 17 is a simple illustration of the VIA Nehemiah and Esther processor pipeline structures (as they affect PadLock ACE). x86 instructions are fetched and translated by the instruction fetch front-end stages (not shown). Internal microinstructions are provided to the execution pipeline from either the x86-instruction translator or the associated microinstruction ROM.

All types of microinstructions proceed through the integer pipeline. Media-type instructions (x87 floating-point, MMX, and SSE instructions) are passed from the E-stage of the integer pipeline to instruction and data queues feeding the appropriate media-execution engines. Once a SSE micro-instruction (for example) is placed in the SSE media queue, it can no longer cause an exception and the integer pipeline and other media units are free to continue executing other instructions while the SSE micro-instruction percolates down the queue. Thus, the execution of XCRYPT instructions (handled as SSE instructions) is overlapped with the execution of subsequent integer, MMX, or floating point instructions.

Figure 17. VIA Execution Pipeline Overview



As an example, referring to the first entry in Table 1, we see that a REP XCRYPTECB instruction requires 31 clocks to execute (for a round count of 10 and ECX=2) from the viewpoint of an SSE instruction that follows the REP XCRYPTECB instruction. If the REP XCRYPTECB instruction is followed by integer, MMX, or floating point instructions, however, the REP XCRYPTECB instruction only appears to take 12 clocks total relative to these instructions. The missing 19 clocks of ACE execution are overlapped with 19 clocks of execution of non-SSE instructions.

In the instruction timing tables below, the following terms are used:

- R = the number of rounds specified
- N = the number of 128-bit blocks to be encrypted

Microcode is optimized for execution with the extended key and control word already loaded. Thus, the ACE is started immediately on the assumption that the system is ready and only then is the EFLAGS bit tested to determine if keys should be loaded. If so, microcode resets ~~the~~ ACE to halt the encryption, loads or generates the key, and restarts. Along the way, ECX is tested for zero. That means that unlike REP string

instructions where an ECX value of zero is a NOP, it is possible for REP XCRYPT instructions to page fault when ECX = 0. When ECX = 0, REP XCRYPT instructions will between 22 and 27 clocks, depending on the state of the ELFAGS:30 bit and the operating mode.

For large values of the REP count we do not provide separate integer timings. The integer pipeline quickly slips and stalls into sync with PadLock ACE. At the completion of the cipher operation, there will be a slight overlap where the integer unit can fetch subsequent instructions while PadLock ACE completes the final block. This overlap is comparable to the extra clocks available when doing a single encryption or decryption.

Table 1: Detailed XCRYPTEBC timing

XCRYPTECB			SSE Clocks		Integer Clocks	
Data Size	**EFLAGS bit 30**	**Extended Key Source**	**Base**	**Variable**	**Base**	**Variable**
128b	1		9	2*R	17	0
128b	0	Hardware	45	2*R	61	0
128b	0	Memory	76	2*R	84	0
2*128b	1		11	2*R	12	0
2*128b	0	Hardware	47	2*R	56	0
2*128b	0	Memory	78	2*R	79	0
N*128b	1		11	N*R		
N*128b	0	Hardware	47	N*R		
N*128b	0	Memory	78	N*R		

Notes

The REP `xcryptecb` instruction always operates on two 128-bit blocks of data. However when the block count in ECX is odd, the extra block of data is not stored.

When ECX is odd and greater than 2, add an extra (R-2) clocks to the SSE Base clocks calculation. The extra R comes from the fact that ACE does an even number of rounds, but saves two clocks on the final store since only the first block is written to memory.

5.2 MEASURED PERFORMANCE

The following data in Table 2 were generated using the sample program provided in Chapter 6 of this manual, running on a 1.00 Ghz VIA Nehemiah processor. When two numbers are given, the second number

shows the performance of a similarly clocked VIA Esther processor. **[NOTE: that these timings for VIA Esther are for first silicon and improvements will be evident in later production steppings.]** Please note that your mileage may vary! The clock() function is not particularly fine-grained and for processes that run for only a few seconds differences in timing measurements on the order of 10% may be observed.

All numbers are in gigabits per second (Gb/s)

Note that for very large data blocks, performance is strictly limited by memory bandwidth as the data loads will almost always miss in the cache.

Table 2: PadLock ACE aligned performance (Nehemiah / Esther)

Data Size	ECB	CBC	CBC-MAC	CFB	OFB
16B	2.81 / 2.56	2.56 / 2.44	- / 2.42	2.56 / 2.27	1.46 / 1.32
32B	5.12 / 4.27	4.11 / 3.76	- / 3.41	4.11 / 3.58	2.05 / 1.95
64B	8.23 / 7.47	5.12 / 4.72	- / 4.37	5.12 / 4.57	2.36 / 2.41
128B	10.23 / 8.81	5.46 / 5.18	- / 4.82	5.46 / 5.15	2.59 / 2.72
256B	11.42 / 9.66	5.61 / 5.36	- / 5.22	5.61 / 5.36	3.61 / 2.92
512B	12.06 / 10.15	5.69 / 5.49	- / 5.39	5.69 / 5.49	2.70 / 3.02
1024B	12.29 / 10.40	5.69 / 5.53	- / 5.51	5.69 / 5.53	2.79 / 3.08
2048B	12.54 / 10.57	5.75 / 5.55	- / 5.55	5.75 / 5.55	2.83 / 3.11
4096B	12.68 / 10.74	5.78 / 5.55	- / 5.55	5.78 / 5.55	2.83 / 3.12
8192B	12.71 / 10.74	5.78 / 5.60	- / 5.55	5.78 / 5.60	2.83 / 3.13
16384B	12.70 / 10.70	5.79 / 5.58	- / 5.58	5.79 / 5.58	2.84 / 3.13
32768B	12.90 / 10.70	5.66 / 5.61	- / 5.58	5.66 / 5.58	2.67 / 3.14
65536B	9.68 / 8.19	5.63 / 5.30	- / 5.30	5.63 / 5.27	2.65 / 2.86
1MB	1.28 / 2.00	1.28 / 1.99	- / 2.19	1.71 / 1.99	0.89 / 1.95

The following data were generated using the sample program provided in Chapter 6 of this manual, running on a 1.00 Ghz VIA Esther processor. Please note that your mileage may vary! The clock() function is not particularly fine-grained and for processes that run for only a few seconds differences in timing measurements on the order of 10% may be observed.

All numbers are in gigabits per second (Gb/s)

Note that for very large data blocks, performance is strictly limited by memory bandwidth as the data loads will almost always miss in the cache.

Please note that the timing numbers here are provisional and based on an early (non-production) stepping of the VIA Esther processor. Production versions of the Esther processor are expected to have nearly double the performance for these operations relative to first silicon. Updated numbers will be provided as soon as they are available.

Table 3: PadLock ACE unaligned performance (Esther)

Data Size	CTR	CBC	CFB	OFB
16B	1.19	0.93	0.91	0.88
32B	1.92	1.13	1.11	1.08
64B	2.23	1.26	1.22	1.20
128B	2.43	1.34	1.33	1.28
256B	2.55	1.38	1.38	1.32
512B	2.61	1.40	1.40	1.34
1024B	2.64	1.42	1.42	1.35
2048B	2.66	1.42	1.42	1.36
4096B	2.66	1.42	1.42	1.36
8192B	2.67	1.42	1.42	1.36
16384B	2.67	1.42	1.42	1.36
32768B	2.67	1.42	1.42	1.36
65536B	2.66	1.37	1.37	1.31
1MB	1.95	1.04	1.04	1.01

CHAPTER

6

SAMPLE CODE

```
/*
  baes.c  --  benchmark/verification for VIA Esther ACE

  Copyright (c) 2004-2005, Centaur Technology, Inc.
  All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

 * Redistributions of source code must retain the above copyright notice,
   this list of conditions and the following disclaimer.

 * Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

 * Neither the name of Centaur Technology nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.

*/

/*
  To build baes:
    gcc -o baes baes.c
```

```
  be familiar with gcc syntax.  So be careful if you optimize code
  using this exact sequence of instructions, as the compiler will
  likely be confused regarding register usage. Also, be careful to
  ensure that buffers are aligned on 16-byte boundaries where so
  required.

  It will be more efficient to create an assembly language subroutine
  that executes the ACE instruction, properly saving and restoring the
  necessary registers.  But the purpose of this source is to illustrate
  how to use the XCRYPT instructions of the PadLock Advanced Cryptography
  Engine, not how to write tight code. Since this program is not optimized,
  the performance numbers generated for PadLock ACE should be treated
  as lower bounds.

  Alternately, experienced gcc programmers can create efficient macros
  that will instruct the compiler regarding register usage by the XCRYPT
  instruction.  In the VIA Programming Guides to the PadLock Random
  Number Generator and the PadLock Hash Engine, examples using such
  macros with the relatively simpler interfaces of the RNG and PHE are
  provided.
*/

#include <stdio.h>
#include <time.h>


/* Various control words*/
#define hardware   0x000
#define key128       0x000 + 0x00a
#define encrypt    0x000
#define decrypt    0x200


int HwkyEncrypt128[4] = {encrypt + hardware + key128, 0, 0, 0};
int HwkyDecrypt128[4] = {decrypt + hardware + key128, 0, 0, 0};


int ecb_monte_e128[4] = {0xAB7743A0, 0xD0B059E2, 0x402DBAB5, 0x1B9701A5};
int ecb_monte_d128[4] = {0x378BBFF5, 0x1F2E6F13, 0x576FEC6B, 0xBAE32120};
int cbc_monte_e128[4] = {0xBF4C842F, 0x0DA7EB78, 0x0196A4A7, 0xB61A8F38};
int cbc_monte_d128[4] = {0x1EB78F9B, 0xF9EF5C03, 0x4613FACB, 0xE0EFACE5};


/* Initial key for timing tests.  We check for functionality, with the
   Monte Carlo test, so we don't really care what we encrypt -- all bits
   look the same, performance-wise */
int ikey[8] = {0x5a5a5a5a};


/* Initialization vector - can be anything for timing */
int ivec[8] = {0};


/* Buffers for monte carlo tests */
```

```
int ctxt[8]  = {0};
int dtxt[8]  = {0};
int zero[8]  = {0};

/* Main encryption buffer */
int buffer_ecb[1024*1024*8];


#define ECB asm (".byte 0xF3, 0x0F, 0xA7, 0xC8\n");
#define CBC asm (".byte 0xF3, 0x0F, 0xA7, 0xD0\n");
#define CTR asm (".byte 0xF3, 0x0F, 0xA7, 0xD8\n");
#define CFB asm (".byte 0xF3, 0x0F, 0xA7, 0xE0\n");
#define OFB asm (".byte 0xF3, 0x0F, 0xA7, 0xE8\n");

#define KEYS  asm ("pushfl\n" \
          "popfl\n");


/* EAX assignment not needed for ECB but makes the macro easier */
#define REGISTERS asm \
   ("leal buffer_ecb, %esi\n" \
    "leal buffer_ecb, %edi\n" \
    "leal ikey, %ebx\n" \
    "leal ivec, %eax\n" \
    "movl buffersize, %ecx\n");

#define ENCRYPT asm  \
   ("leal HwkyEncrypt128, %edx\n"); \
  REGISTERS

#define DECRYPT asm  \
   ("leal HwkyEncrypt128, %edx\n"); \
  REGISTERS

#define MISALIGN asm \
   ("incl %esi\n" \
    "incl %edi\n");

#define UNALIGNED 0x20
#define MAC_MODE  0x10
```

```
int loopcounter = 0;
int buffersize = 0;

void reset_buffers() {
  int i;

  for (i=0; i<8; i++) {
    zero[i] = 0;
    dtxt[i] = 0;
    ctxt[i] = 0;
  }
}

int monte_carlo_cbc() {

  int j, k, n;

  reset_buffers();
  for (k=0; k < 400; k++) {

    KEYS
    for (j=0; j < 10000; j++) {

      asm ("leal HwkyEncrypt128, %edx\n");
      asm ("leal ctxt, %esi\n");
      asm ("leal ctxt + 16, %edi\n");
      asm ("leal dtxt, %ebx\n");
      asm ("leal zero, %eax\n");
      asm ("movl $1, %ecx\n");
      CBC

      for (n=0; n<4; n++) {
        ctxt[n] = zero[n];
        zero[n] = ctxt[4+n];
      }
    }

    for (n=0; n<4; n++)
      dtxt[n] = dtxt[n] ^ zero[n];
```

```
}

for (n=0; n<4; n++) {
  if (zero[n] != cbc_monte_e128[n]) {
    printf("CBC 128BIT ENCRYPT MONTE CARLO FAILED\n");
    exit(0);
  }
}

reset_buffers();
for (k=0; k < 400; k++) {

  KEYS
  for (j=0; j < 10000; j++) {

    asm ("leal HwkyDecrypt128, %edx\n");
    asm ("leal ctxt, %esi\n");
    asm ("leal ctxt + 16, %edi\n");
    asm ("leal dtxt, %ebx\n");
    asm ("leal zero, %eax\n");
    asm ("movl $1, %ecx\n");
    CBC

    for (n=0; n<4; n++) {
      zero[n] = ctxt[n];
      ctxt[n] = ctxt[4+n];
    }
  }

  for (j=0; j<4; j++)
    dtxt[j] = dtxt[j] ^ ctxt[j];

}

for (j=0; j<4; j++) {
  if (ctxt[j] != cbc_monte_d128[j]) {
    printf("CBC 128BIT DECRYPT MONTE CARLO FAILED\n");
    exit(0);
  }
}
```

```
}

int process_command_line(int argc, char **argv) {

        int i;
        char * c;
  int retval = 0;

        for (i = 1; i < argc; i++)  {

                c = argv[i];

                if (!strcmp(c,"-rep"))
                        loopcounter = atoi(argv[++i]);

                if (!strcmp(c,"-buffer"))
                        buffersize = atoi(argv[++i]);

     if (!strcmp(c,"-mc"))
       retval = 1;
  }

/* Default the encryption buffer to 8K */
  if (buffersize == 0 )
     buffersize = 512;

/* And don't let it get bigger than 32M */
  if (buffersize > 1024*32*64)
     buffersize = 1024*32*64;

/* Set the hardware default if the user hasn't specified a loop count */
  if (loopcounter == 0)
     loopcounter = 100000;

  return retval;
}


int main(int argc, char **argv) {
```

```
  clock_t start;
  int j;
  float datarate;

/* Check the command line to possibly tweak the operation */
        if (process_command_line(argc, argv)) {
    printf("\nRunning Monte Carlo tests ...\n");
    monte_carlo_cbc();
    printf("PASSED\n");
  }

  printf("\nEncrypting %d byte block %d times.\n\n", buffersize * 16, loop-
counter);

#define TIME(MODE,METHOD,TEXT) \
  start = clock(); \
  for (j=0; j < loopcounter; j++) { \
    METHOD; \
    MODE; \
  } \
  datarate = (float) CLOCKS_PER_SEC * j * buffersize * 128. / (1000000000. *
(float) (clock() - start)); \
  printf(TEXT); \
  printf(" rate: %.2f Gbs\n\n", datarate); \

#define MTIME(MODE,METHOD,TEXT) \
  start = clock(); \
  for (j=0; j < loopcounter; j++) { \
    METHOD; \
    MISALIGN; \
    MODE; \
  } \
  datarate = (float) CLOCKS_PER_SEC * j * buffersize * 128. / (1000000000. *
(float) (clock() - start)); \
  printf(TEXT); \
  printf(" rate: %.2f Gbs\n\n", datarate); \

  TIME(CBC,ENCRYPT,"CBC mode encrypt")
  TIME(CBC,DECRYPT,"CBC mode decrypt")
  TIME(ECB,ENCRYPT,"ECB mode encrypt")
  TIME(ECB,DECRYPT,"ECB mode decrypt")
```

```
  TIME(CFB,ENCRYPT,"CFB mode encrypt")
  TIME(CFB,DECRYPT,"CFB mode decrypt")
  TIME(OFB,ENCRYPT,"OFB mode encrypt")
  TIME(OFB,DECRYPT,"OFB mode decrypt")
  TIME(CTR,ENCRYPT,"CTR mode encrypt")
  TIME(CTR,DECRYPT,"CTR mode decrypt")

/* Set the control words for unaligned operation */
  HwkyEncrypt128[0] += UNALIGNED;
  HwkyDecrypt128[0] += UNALIGNED;

  TIME(CBC,ENCRYPT,"CBC mode encrypt (unaligned)")
  TIME(CBC,DECRYPT,"CBC mode decrypt (unaligned)")
  TIME(CFB,ENCRYPT,"CFB mode encrypt (unaligned)")
  TIME(CFB,DECRYPT,"CFB mode decrypt (unaligned)")
  TIME(OFB,ENCRYPT,"OFB mode encrypt (unaligned)")
  TIME(OFB,DECRYPT,"OFB mode decrypt (unaligned)")
  TIME(CTR,ENCRYPT,"CTR mode encrypt (unaligned)")
  TIME(CTR,DECRYPT,"CTR mode decrypt (unaligned)")

/* Reset control words for aligned MAC */
  HwkyEncrypt128[0] += MAC_MODE - UNALIGNED;

  TIME(CBC,ENCRYPT,"CBC-MAC mode")
}
```